To appear in the proceedings of the 5th ACM SIGGRAPH Symposium on Solid Modeling and Applications

A Coherent Sweep Plane Slicer for Layered Manufacturing

Sara McMains Carlo Séquin

Computer Science Department
University of California, Berkeley *

Abstract

We describe the design and implementation of a coherent sweep plane slicer, built on top of a topological data structure, which "slices" a tessellated 3-D CAD model into horizontal, 2.5-D layers of uniform thickness for input to layered manufacturing processes. Previous algorithms for slicing a 3-D b-rep into the layers that form the process plan for these machines have treated each slice operation as an individual intersection with a plane, which is needlessly inefficient given the significant coherence between the finely spaced slices. An additional shortcoming of many existing slicers that we address is a lack of robustness when dealing with non-manifold geometry. Our algorithm exploits both geometric and topological inter-slice coherence to output clean slices with explicit nesting of contours.

Keywords: rapid prototyping, computational geometry, topology, slicing, .STL format, CAD/CAM

1 Introduction

Designers who want to make prototypes of solid three-dimensional parts directly from CAD descriptions are increasingly turning to a class of technologies collectively referred to as layered manufacturing or solid freeform fabrication (SFF). These technologies include stereolithography, 3-D printing, fused deposition modeling, selective laser sintering, and laminated object manufacturing [4]. In all these processes, a triangulated boundary representation (b-rep) of the CAD model of the part in .STL format [1] is sliced into horizontal, 2.5-D layers of uniform thickness. Each cross sectional layer is successively deposited, hardened, fused, or cut, depending on the particular process, and attached to the layer beneath it. The stacked layers form the final part.

Previous algorithms for slicing a 3-D b-rep into the layers that form the process plan for these machines have treated each slice operation as an individual intersection with a plane. But for a typical stereolithography build with .005" layers, a 5" high part will be made from one thousand parallel slices with significant coherence between slices, which can be used to calculate neighboring slices more efficently. An additional shortcoming of many existing slicers is a lack of robustness when dealing with non-manifold geometry.

We describe a new algorithm for slicing a polyhedral solid model with simple convex faces, such as an .STL file. Our algorithm exploits both geometric and topological inter-slice coherence to output clean slices with no self-intersections and explicit nesting of contours. First we derive the connectivity of the b-rep (this is necessary in any case to validate that the input describes a closed solid) and build a variant of the radial edge structure that records this topological information. We also logically separate pseudo-2-manifold edges and vertices into coincident manifold edges and vertices, so that they will be handled correctly. The main body of the algorithm employs a sweep plane approach, using the connectivity information for the 3-D solid to derive and update the connectivity of subsequent 2-D slices. Actual intersection calculations on the edges are performed quickly and efficiently through incremental updates between slices. The resulting slice descriptions are topologically consistent, connected, nested contours, rather than unordered collections of edges.

* {sara | sequin}@cs.berkeley.edu

2 Prior Work

2.1 Representation

The internal data structure that we build is closely related to Weiler's radial edge structure [19]. The radial edge structure is a generalization of Baumgart's winged edge data structure [3] to non-manifold geometry. These data structures allow us to answer questions about topological adjacency relationships, e.g. the faces incident to an edge, often either in constant time or in time proportional to the size of the output set. Weiler's data structure also records the radial ordering of faces around non-manifold edges (hence its name). Another important concept from Weiler's work is the distinction between an abstract, unoriented geometric entity, such as an edge, and an oriented use of that entity, such as a directed *edge_use* that describes part of the boundary of a face. Other variations on these data structures include Mäntylä's half-edge data structure [13], which is limited to 2-manifolds, Rock and Wozny's topological data structure for .STL, also limited to 2-manifolds, and the data structure that ACIS modelers build and exchange in .sat files [17].

Another pioneering non-manifold representation forms the basis for the Noodles system developed by Gursoz, et al [9]. In addition to recording the radial ordering of faces around non-manifold edges, Noodles records the relationships between the distinct regions of space around non-manifold vertices. These regions are called *zones*. The manifold sheets that form the boundaries between zones are called *disks* (see figure 1).

Each disk in the neighborhood of a vertex records a cyclically ordered list of edge_uses that are both incident to the vertex and on the surface of the disk. This is called a disk cycle (see figure 2). Our slicing algorithm processes each disk cycle separately at pseudo-2-manifold vertices.

2.2 Slicing

One of the simplest slicing algorithms for .STL files intersects all triangles with each z-plane and connects the resulting line segments into closed polygons, one slice at a time. This is the approach that is used by Kirschman, et al [10], who also parallelized this algorithm.

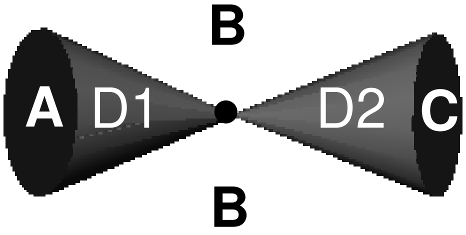

Figure 1: The vertex in the center is surrounded by three 3-D zones: A and C, the regions inside the cones, and B, the region outside the cones. The zones are bounded by two 2-D disks, D1 and D2.

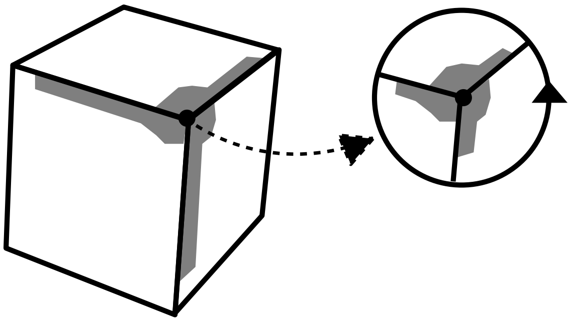

Figure 2: This vertex has a single disk cycle made up of three edge_uses. The edge_uses are depicted as shaded regions on the face_uses that they bound (figure adapted from Gursoz et al).

Mäntylä [13] developed a "splitting algorithm," which uses his half-edge data structure. His algorithm, like ours, is topology based, but since he is making only a single slice through the part, the technique is quite different; there is no coherence to exploit. Another difference is that he builds and outputs solid models (one on each side of the slice plane), not 2.5-D slices. Rock and Wozny [15] also build a topological model before slicing and use it to derive connectivity on a slice by slice basis by marching from one intersection face to the next based on their connectivity. To find multiple contours in a single slice, they search for unvisited edges. They don't exploit topological coherence between slices, but they do use geometric coherence to calculate intersection coordinates incrementally. Dolenc and Mäkelä [7] and Kulkarni and Dutta [11] concentrate on where to slice a part to obtain the greatest accuracy with the fewest number of 2.5-D slices when adaptive slice thicknesses are an option. Our algorithm is equally suitable for adaptive or for uniform slice thicknesses.

3 Verification and Preprocessing

3.1 The Topological Data Structure

We need a solid representation scheme that will support all bounded, rigid solids, or *r-sets* [18], including not only 2-manifold solids but also the subset of non-manifold geometry that corresponds to physically realizable solid objects. We call such geometry *pseudo-2-manifold*, after Mäntylä [12]. On a 2-manifold boundary, the mathematical neighborhood of each point is topologically equivalent to a 2-D disk. Any geometry that doesn't have this property is *non-manifold*. In the b-rep of a 2-manifold, each edge is used exactly twice in opposite directions (a necessary but not a sufficient condition for manifoldness). The set of pseudo-2-manifold objects is a subset of non-manifold objects. On a pseudo-2-manifold boundary, the neighborhood of each point is topologically equivalent to n disks, $n > 2$, and each edge in the b-rep is used an equal number of times in both directions.

The first step in our algorithm is to build our variant of the radial edge structure. For this application, we have stripped away some of the overhead that we don't require from Weiler's radial edge structure. Weiler separates undirected loops or faces from directed face_uses and loop_uses, allowing the same face to be referenced from both sides where it forms a membrane between cells, for example. We only represent the actual directed face_uses and loop_uses, since a single face or loop is unlikely to be used more than once in SFF file descriptions. For simplicity, we will refer to face_uses and loop_uses as faces and loops in the rest of this paper.

Each face (see table 1) is defined by one counter-clockwise, outer loop and a (possibly empty) list of clockwise, inner hole loops. For triangulated .STL input, we will have no inner hole loops in the input geometry. Loops are implicit: in a face, we store a loop simply as a pointer to an arbitrarily chosen edge_use in that loop. Each edge_use stores a pointer to the next edge_use in the loop; we follow these pointers to traverse the loop.

FACE	
EDGE_USE *	First_Outer_Loop_Edge_Use
List<EDGE_USE *>	Inner_Loop_List
VOID *	Extra

Table 1: The member data for a face.

Each edge_use in the loop points back to the face whose boundary it helps to define (see table 2). To make edge_uses compact, we have chosen to store only one vertex pointer with each edge_use, a pointer to the root vertex (the vertex from which the edge_use is directed away). The vertex on the other end can be found by following the pointer to the next edge_use in the loop and getting its root vertex. While we represent each edge_use explicitly, the abstract, undirected edges are represented implicitly by circular lists of edge_uses sharing the same endpoints, linked by "sibling edge_use" pointers stored with each edge_use. We use a hash table of pairs of pointers to the vertices of the edge_use to match up sibling pointers initially. Later, they are sorted radially.

EDGE_USE	
FACE *	Face
VERTEX *	Root_Vertex
EDGE_USE *	Next_In_Loop_Edge_Use
EDGE_USE *	Sibling_Edge_Use
EDGE_USE *	Next_Vertex_Edge_Use
VOID *	Extra

Table 2: The member data for an edge_use. The Next_Vertex_Edge_Use field is explained below.

An important feature of our data structure is constant space storage for each vertex and for each edge_use. Rather than storing a variable length list of all of the edge_uses incident to a vertex with the vertex, we chain together all of the vertex's edge_uses into a circular linked list (in arbitrary order) via the Next_Vertex_Edge_Use field in the edge_uses instead. The vertex contains a pointer to any one of these edge_uses (the First_Vertex_Edge_Use field in table 3). The combination of these pointers allows us to iterate through all of the vertex's edge_uses, even at non-manifold vertices. To build this part of the data structure from .STL input, we construct a hash table on the vertex coordinates to match up vertices shared by edge_uses in adjacent triangles.

For representing .STL input, the space usage for each face is also constant since the faces have no inner hole loops. Constant space

VERTEX	
double	X
double	Y
double	Z
double	W
EDGE_USE *	First_Vertex_Edge_Use
VOID *	Extra

Table 3: The member data for a vertex. We store a homogeneous vertex coordinate for compatibility with our matrix libraries.

storage is important for allocating memory efficiently; it allows us to pre-allocate storage in arrays.

3.2 Input Verification & Cleanup

Our next step is to verify that the input describes a closed solid. As a first check, we confirm that every edge is used an equal number of times in each direction. This ensures that the input doesn't contain any cracks or T-junctions. Another possible verification step is to check that none of the input faces intersect each other, or we could get intersecting or self-intersecting slice contours, which some machines may not process gracefully (though the software we have tested for SLS and Zcorp machines takes unions of intersecting geometry). We will be adding this check to our system in the future.

If the solid is not closed, we do some limited automated cleanup. We identify the end points of all non-manifold, non-pseudo-2-manifold edges, and merge any that are closer than the shortest edge seen or some user defined epsilon. If the problems are more serious than roundoff errors, a number of commercial or research systems ([5, 2, 14]) can be used to fix the file.

3.3 Separating Pseudo-2-Manifold Edges

The next step is to separate any pseudo-2-manifold edges, such as the one pictured in figure 3, to get coincident 2-manifold edges in the topological data structure. We do this by matching the edge_uses into pairs with mutually referring sibling pointers.

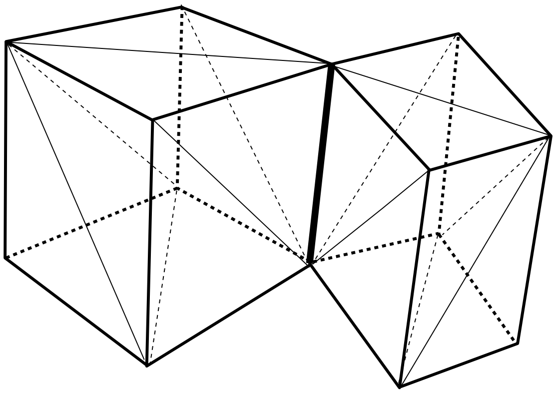

Figure 3: Geometry with a pseudo-2-manifold edge (the bold edge in the center) that must be separated. The lightest lines indicate boundaries between coplanar triangles.

First we perform a radial sort of the faces incident to the edge. If we project the faces onto a plane perpendicular to the edge (see figure 4) and look at the corresponding projected face normals (see figure 5), the normals should alternate between being directed clockwise and counter-clockwise around the edge. (If they do not, it indicates that the geometry was self-intersecting, and we reject the input.) The normals point to the side of the face that is empty space, away from the side that contains material (the interior of the part).

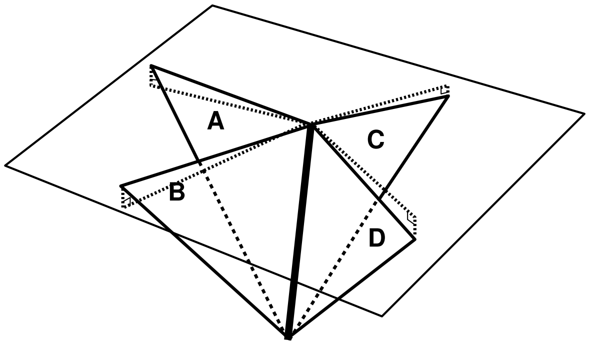

Figure 4: The faces incident to the pseudo-2-manifold edge, and the projection plane for the edge.

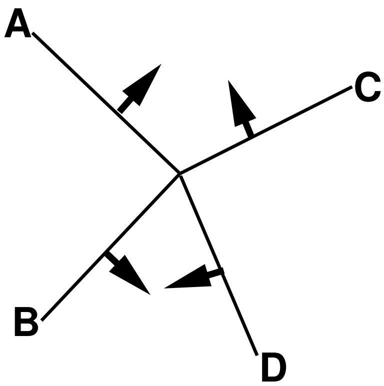

Figure 5: The projected faces with their projected normals. A and D have clockwise normals; B and C have counterclockwise normals.

We need to choose pairs of faces whose edge_uses along the edge in question are in opposite directions in order to make separate 2-manifold edges. There are two obvious choices for this example that won't form intersections in the boundary: either we could pair adjacent faces whose normals point away from each other (e.g. A & B, C & D), or faces whose normals point towards each other (e. g. B & D, A & C). The first alternative means that in a slice through the separated edges, the surrounding contour(s) will bound separate regions of material that happen to touch at this edge, and the second alternative means that in a slice through the separated edges, the surrounding contour will bound the same region of material that just happens to pinch down to zero thickness at this edge (see figures 6 and 7). If more than four edge_uses are coincident, combinations of these two choices are also possible.

We have chosen to match the faces into pairs of adjacent faces with normals pointing away from each other, but this choice is somewhat arbitrary The end result from the point of view of the edge_uses is separate but coincident 2-manifold edges that have the same vertices as endpoints, each with exactly two edge_uses in opposite directions with mutually referencing sibling pointers. While this matching scheme doesn't give us full symmetry between full and empty regions, it does give us a globally consistent boundary. For example, if the two touching cubes were attached by a solid pedestal underneath, then after the pseudo-2-manifold edge separation, the vertex at the top of the edge would have two separate disk cycles around the two cubes, while the vertex at the bottom would

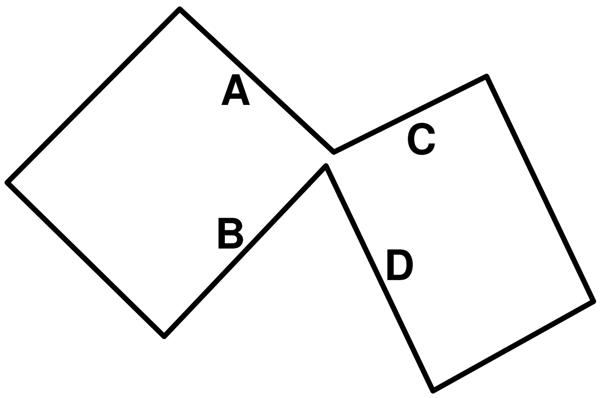

Figure 6: Matching faces B & D, A & C, whose normals point toward each other, is topologically equivalent to the geometry pictured here.

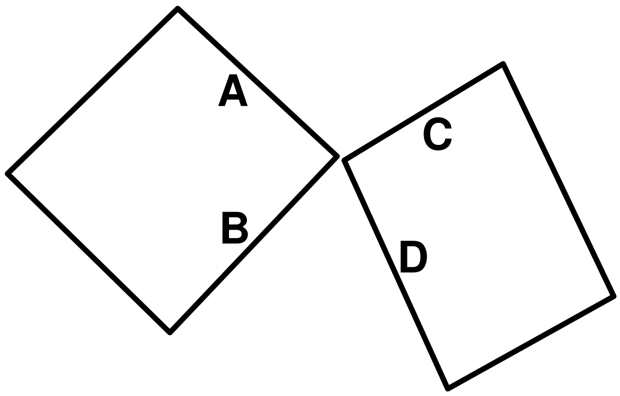

Figure 7: Matching faces A & B, C & D, whose normals point away from each other, is topologically equivalent to the geometry pictured here.

have a single disk cycle connected by the faces of the pedestal. This configuration is still consistent even though it is not symmetric.

4 Sweep Plane Algorithm

4.1 Overview

The general approach in a sweep plane algorithm involves moving a virtual horizontal plane from bottom to top over the input. While the plane moves along, the algorithm maintains a dynamic *status structure*, generally a parameterized representation of the intersection of the input with the current position of the sweep plane. This status information only changes at certain points called *event points*. Whenever the sweep plane reaches an event point, the point is processed to update the status structure, and additional calculations, depending on the particular problem to be solved, are performed. A desirable feature of many sweep algorithms is that their running times increase monotonically with the amount of output produced. For example, the number of intersections of n line segments in the plane could be as many as n^2, but a sweep line algorithm to find them has running time $O((n+k)logn)$, where k is the number of intersections discovered [6].

In our algorithm, we move our virtual sweep plane from the bottom to the top of the input, the direction in which SFF machines typically build parts. For the discussion here, we assume that we are slicing perpendicular to the z-axis, so we will process the vertices from least to greatest z-coordinate. The algorithm's "z-events" occur each time the sweep plane hits a vertex (an event that changes the topology of the current slice) and each time the sweep plane hits a height at which we wish to output a slice (an event where we wish to derive the geometry of the current slice).

For a given position of the sweep plane, after we have processed any new vertices that intersect that plane, our status structure will reflect the topology of a slice – the connectivity of its vertices – at the current height, recorded in a list of contours. For each slice contour in the list, the data structure will contain a circular, ordered, doubly-linked list of the edges of the solid intersected by the current sweep plane that determine this contour (see figure 8). An outer contour also points to a list of any inner hole contours nested within it, and an inner hole contour points to the outer contour that contains it.

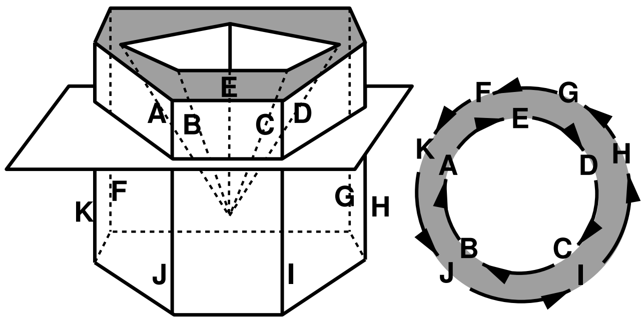

Figure 8: The slice data structure at the given position of the sweep plane will consist of the two circular linked lists pictured at right. Their orientation and nesting indicate that the slice contains a single polygon with a hole.

Each of the intersected edges of the solid contained in the slice contour structure determines a vertex in the 2-D slice, with the same connectivity as in the contours' circular list(s) of edges. The geometric positions of the vertices are derived by evaluating an intersection point at the current z-height on each edge in the circular list. Each contour's clockwise or counter-clockwise orientation (from the point of view of looking down on the slices) determines whether it is an inner hole contour or an outer contour, respectively. Because we don't need this orientation classification until we output a slice, we wait until we get to the first slice after the contour is formed to classify it, since we will be calculating intersection point coordinates then anyhow, and those geometric positions are needed for deriving the orientation. In addition to contour orientation, we also derive the nesting of contours inside one another at the first slice after their creation. The contour orientations as well as their nesting will remain unchanged until we hit a saddle point (assuming that the input is not self-intersecting), at which point we mark the orientation and nesting invalid and rederive them at the next slice (see sections 4.2.4 and 4.3).

The algorithm proceeds by moving the plane up to the next z-event. If there are any new vertices at this height, we process them in arbitrary order, using information about the incident edges to modify the appropriate contours, as explained in detail below. If we want to output a slice at this height, then (after first processing any new vertices) we calculate the (x,y) coordinates of the intersections with the edges, derive orientation and nesting for any new or changed contours, and then output the geometry as well as the topology of the slice.

4.2 Vertex Processing

The topological structure that we build doesn't include disk cycles, so we must find them as the first step in vertex processing. For our algorithm, for each manifold sheet (disk) incident to the vertex, we want a clockwise cycle (as seen from the exterior of the part) of the edge_uses on that disk rooted at that vertex (see figure 9).

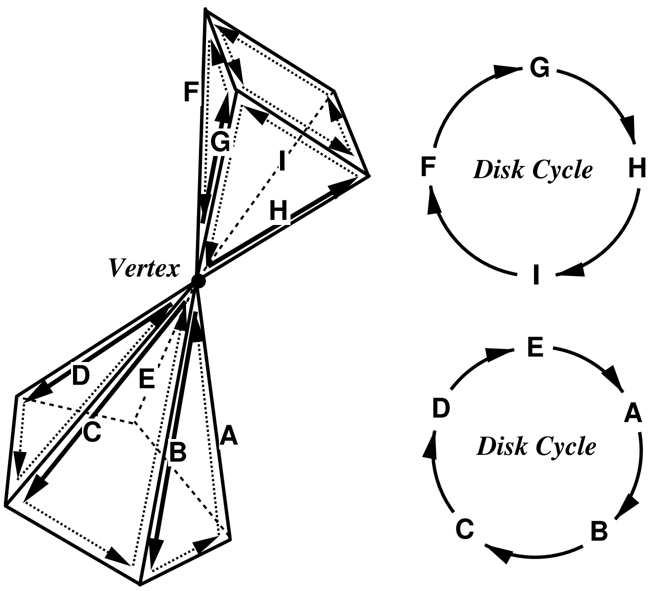

Figure 9: A pseudo-2-manifold vertex with two disk cycles. The edge_uses are shown as arrows on the visible faces; those that appear in the disk cycles for the center vertex are solid and those that do not are dotted. The disk cycles are pictured on the right.

These are the same as the Noodles disk cycles (see figure 2) except that we want them oriented clockwise instead of counterclockwise for compatibility with our slice data structure. The disk cycle is stored as a circular, doubly-linked list of edge_uses, pieces of which will be incorporated directly into slice contour lists, also doubly linked (though all figures show only the "next" arrows of these doubly linked lists.) To build a disk cycle, we take the vertex's first edge_use (B in figure 10), insert it as the first edge_use in our linked list for this disk cycle, then find its sibling's next-in-loop pointer and insert this edge_use (C in figure 10) as the next clockwise edge_use in the disk cycle, and so on. When we're back to the first edge_use we inserted, we've completed a disk cycle and we close the list into a circular linked list. The pseudo-code follows:

```
First_Edge_Use = Cur_Edge_Use =
  Vertex->First_Edge_Use;
do {
  Prev_Edge_Use = Cur_Edge_Use;
  Cur_Edge_Use = Prev_Edge_Use->Sib-
    ling_Edge_Use->Next_In_Loop_Edge_Use;
  Disk_Cycle->Insert_Next(Cur_Edge_Use);
} while (Cur_Edge_Use != First_Edge_Use);
```

Next, we check if there are more disk cycles around the vertex by following its first edge_use and iterating through the next vertex edge_use pointers. If we come to an edge_use that has not yet been included in a disk cycle (such as H in figure 10), then we know that this is a non-manifold vertex. We build the next disk cycle in the same manner as the first, starting from this edge_use. We continue until all of the edge_uses rooted at the vertex have been included in a disk cycle.

Next we process the disk cycle(s) we have constructed for the vertex. If there are multiple disk cycles, we process each in turn;

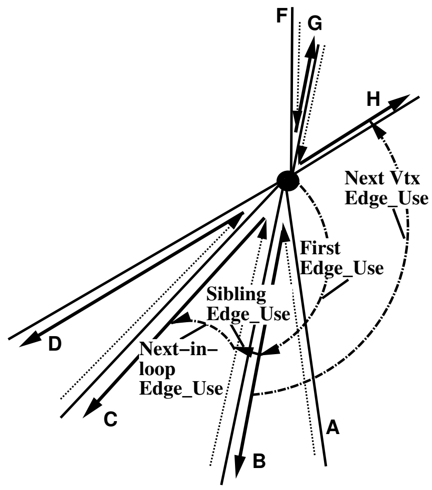

Figure 10: Following pointers to build disk cycles.

the processing order doesn't affect the results. Each edge_use in the disk cycle is classified as one of two types. Either we have already processed its sibling (and its sibling is in one of our existing slice contours), in which case this vertex is the top of an edge_use which won't appear in future slices and we refer to it as an *ending edge_use*, or it will be an edge_use whose sibling we haven't seen before, in which case this vertex is the bottom of a new edge_use and we refer to it as a *beginning edge_use*. For the vertex in the example, A, B, C, D, and E are ending edge_uses, while F, G, H, and I are beginning edge_uses. For a horizontal edge, the disk cycle for the first endpoint we process will treat the horizontal edge_use as a beginning edge_use, and the disk cycle at the other endpoint will treat its sibling as an ending edge_use. No special case is required for horizontal edges, nor does the order in which we process their endpoints affect the output.

For the first vertex we process, and again at local minima on the part, the disk cycle will consist entirely of beginning edge_uses. For the final vertex and at local maxima, the disk cycle will consist entirely of ending edge_uses. The majority of disk cycles will consist of a mix of beginning and ending edge_uses. We consider each of these three cases separately.

4.2.1 Beginning Vertices

A disk cycle with all beginning edge_uses is converted directly to a new contour that will appear in slices above the vertex. For the first vertex processed, the vertex at the bottom-most point of the part, its disk cycle, with its clockwise ordering of edge_uses around the vertex as seen from the outside of the part, gives a counter-clockwise ordering of the edge_uses in the slice contour viewed from the top of the part. This counter-clockwise ordering indicates an outer contour in the slice (see figure 11). This same situation is encountered at any local minimum for geometry with convex faces only. For the case shown in figure 12, the clockwise ordering of edge_uses in the disk cycle corresponds to a clockwise ordering in the corresponding slice contour viewed from the top of the part, indicating a hole in the slice. In either case, the disk cycle becomes a slice contour; we add it to the list of contours and update all of its edge_uses to point to the slice contour that they now belong to (making use of the general data structure's Extra pointer). The slice contour pointer also serves to flag whether an edge has been seen before; to tell if an edge_use is an ending edge_use we check to see if its sibling's slice contour pointer is set. We don't try to determine if our new slice contour is an outer contour or a hole contour until we reach a height where we need to output an actual slice.

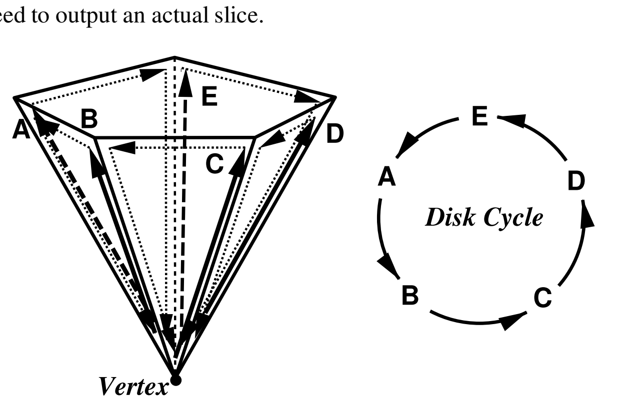

Figure 11: The disk cycle around the indicated vertex contains all beginning edge_uses; it starts a new counterclockwise outer contour in the slice data structure (seen from above).

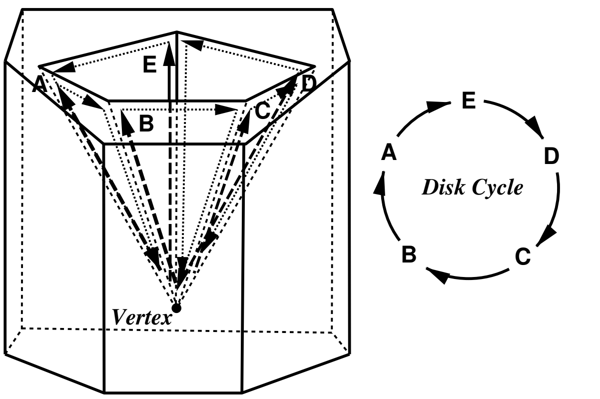

Figure 12: The disk cycle around the indicated vertex has the opposite orientation as the previous example; it starts a new clockwise inner hole contour in the slice data structure (seen from above).

4.2.2 Ending Vertices

A vertex with all ending edge_uses (we will use a prime (') on edge_use labels in subsequent figures to indicate ending edge_uses) is the analogue of the beginning vertex case. It arises when an existing slice contour will disappear from all slices above this vertex. We find the existing slice contour that it matches, which could be either an outer contour or a hole (see figures 13 and 14), and delete it from the list of contours. To find the matching existing contour, we look at the slice contour pointer of the sibling of any of the ending edge_uses in the disk cycle; this existing contour will contain the siblings of these ending edge_uses ordered in the opposite sense (clockwise or counterclockwise).

When we delete a contour, it has shrunk to zero area, so there should be no other contours contained in it (except for the case of a pseudo-2-manifold vertex where any contained contours will disappear at the same vertex after we process their disk cycles). A deleted contour at any vertex may, however, have been contained in another contour, and if so we delete it from that contour's list of contained contours.

4.2.3 Mixed Vertices

When we have both beginning and ending edge_uses in a disk cycle, we use the ending edge_uses to determine where to add the beginning edge_uses to the existing slice contours. First let's consider the simple case illustrated in figure 15, where we start with the slice contour shown at the bottom right, and end up with the contour shown at the top right.

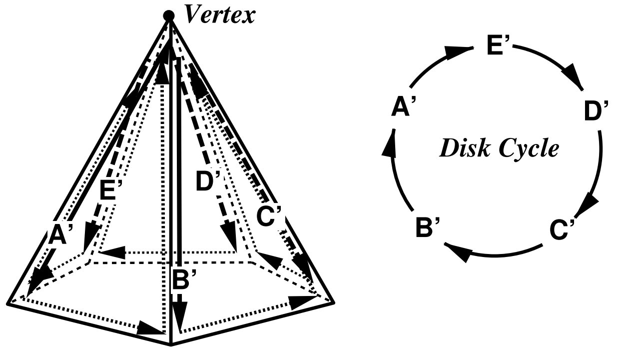

Figure 13: An outer contour is deleted at this vertex where all the edge_uses are ending edge_uses.

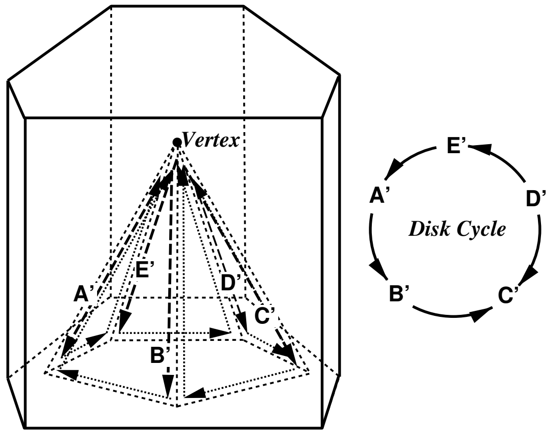

Figure 14: An inner hole contour is deleted at this vertex where all the edge_uses are ending edge_uses.

Below the center vertex, we have the existing slice contour (A–>B–>C–>D–>G–>). At the center vertex, we process the disk cycle (C'–>B'–>E–>F–>) (see figure 16). This disk cycle contains one *ending run*, (C'–>B'), which is a run of consecutive ending edge_uses whose siblings match up with a run of consecutive edge_uses in the existing slice contour (the matching siblings will be in the opposite order, B–>C). The remaining edge_uses in the disk cycle, (E–>F), will be replacing the matching siblings to form the new slice contour.

Pointers from the first edge_use in the ending run, (C'), provide us with the information needed to splice between (D) in the existing slice contour and (F) in the disk cycle. We take (C')'s predecessor in the disk cycle, (F), and have it point to (D), the edge_use following (C')'s sibling in the existing slice contour. The pseudo-code for this operation follows, where FirstEnd refers to the first ending edge_use in the ending run in the disk cycle:

```
FirstEnd->Previous->Next =
  FirstEnd->Sibling->Next;
```

Similarly, we follow pointers from the last edge_use in the ending run, (B'), to splice between (A) in the existing slice contour and (E)

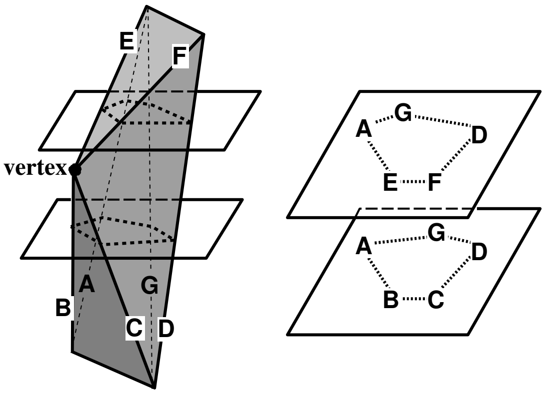

Figure 15: We look at how processing the disk cycle at the center vertex changes the slice contours pictured at right.

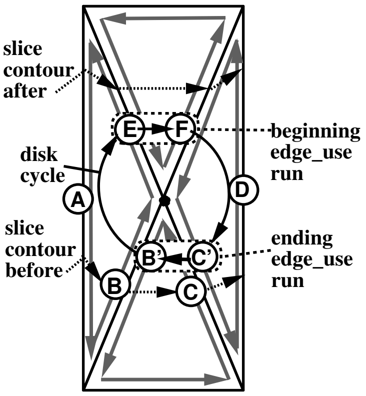

Figure 16: A front view of the same geometry showing the connectivity of the disk cycle in the center (solid black arrows) and the relevant connectivity of the slice contour (dotted arrows) before (below) and after (above) processing the disk cycle. Ending edge_uses are denoted with a prime ('); their beginning edge_use siblings have the same label without the prime. The ending edge_use run and beginning edge_use run are demarcated with dashed ovals.

in the disk cycle. We take (B')'s sibling's predecessor in the existing slice contour, (A), and have it point to (E) instead, the edge_use following (B') in the disk cycle. The pseudo-code for this operation follows, where LastEnd refers to the last ending edge_use in the ending run in the disk cycle:

```
LastEnd->Sibling->Previous->Next =
  LastEnd->Next;
```

The piece of the linked list between the first and last edge_uses in the beginning edge_use run, (E–>F) in this case, is retained from the disk cycle and incorporated directly into the updated slice contour. In addition, we make the complementary changes to the corresponding "previous" pointers to maintain the doubly linked lists:

```
FirstEnd->Sibling->Next->Previous =
  FirstEnd->Previous;
LastEnd->Next->Previous =
  LastEnd->Sibling->Previous;
```

The final step is to traverse the beginning edge_use run from the disk cycle, (E–>F), and store with each beginning edge_use a pointer to the slice contour to which it has been added.

4.2.4 Saddle vertices

For the simple case of a convex part, there is only a single run of ending edge_uses in each disk cycle, and we are just replacing this run in the corresponding slice contour with a new run of beginning edge_uses. The beginning edge_uses all get added to the same contour that the ending run was in, and no other edge_uses or contours are affected. For more complicated geometry such as the three pronged part shown in figure 17, we may encounter multiple runs of ending edge_uses in a single disk cycle (see figures 18 and 19). When this occurs, we process the ending runs sequentially. The splicing code is identical to the code for a single end run, but instead of just substituting new edge_uses into a single spot in an existing slice contour, the splices may split apart or merge together existing slice contours; thus the updating of the contour pointers will require more attention.

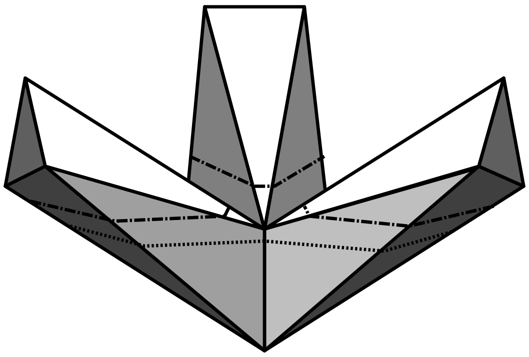

Figure 17: In this side view of a three pronged part, there is a single contour per slice below the center vertex, as indicated by the lower dotted line, and three contours per slice above the center vertex, as indicated by the upper dashed lines.

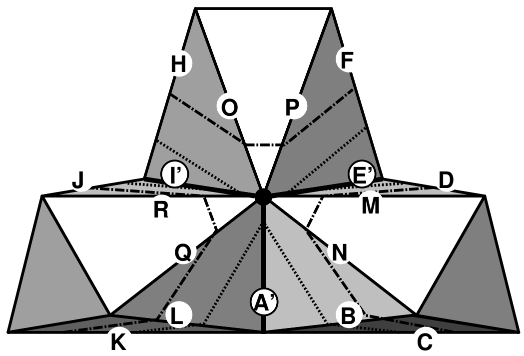

Figure 18: Looking down on this same part, we see the center vertex whose disk cycle has three ending edge_uses (the bold edge_uses with black circles around their primed labels). Each of the ending "runs" is one of these edge_uses.

In the example shown, a single contour splits into three contours after the three ending runs are processed for the vertex at the center saddle point (see figures 20, 21, and 22). If the same part was upside down, we would have three slice contours merging into one at this vertex.

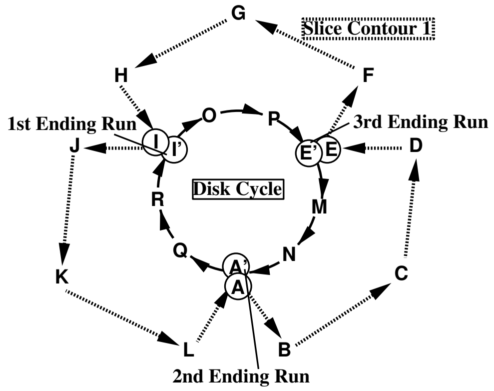

Figure 19: The disk cycle and existing slice contour before we process the center vertex. Ending edge_uses are denoted with a prime (').

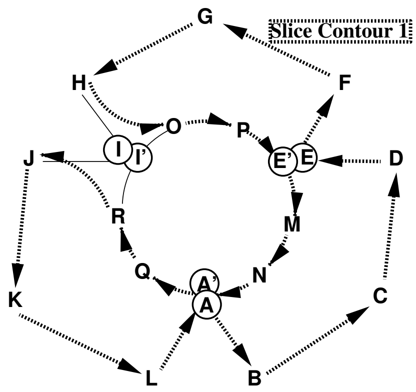

Figure 20: After splicing around the first ending run, the disk cycle is merged with the existing slice contour.

When such splitting or merging occurs, we need to identify which contours have changed, update their edge_uses to point to the contour they now belong to, and update the list of slice contours. We determine if a pair of splices has caused a split or a merge by comparing the old slice contour pointers of either pair of edge_uses that we've spliced together (e.g. J and R, the pair spliced around the first edge_use in the first ending run in this example). If they had different slice contour pointers before we connected them, then the pair of splices merges these two contours together (see figure 20). If they had the same slice contour pointer before we connected them, then the pair of splices splits that contour (see figure 21). In either case, in one of the two contours that merged or that resulted from the split, we need to reset the pointer from each edge_use to its new slice contour. We process each ending run, performing its corresponding pair of splicing operations and updating all the edge_uses' slice contour pointers, based on the connectivity that resulted from the previous pair of splices, before processing the next ending run.

The pair of splices for the first ending run in a disk cycle always merges the new disk cycle with an existing slice contour; all of the remaining unprocessed edge_uses of the disk cycle have their slice contour pointers set to this slice contour (as in figure 20). For the simple case of a single ending run described in the previous section, this is the end of processing. After this pair of splices, subsequent pairs of splices generated by the same disk cycle may cause splits, if other ending runs in the disk cycle match up with edge_uses in this same slice contour (as in figures 21 and 22), or they may cause further merges, such as would happen at this vertex if the same part

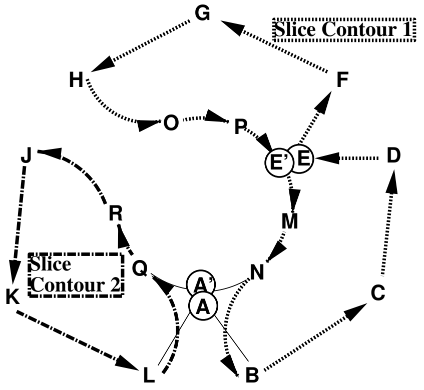

Figure 21: After splicing around the second ending run, the slice contour splits into two pieces.

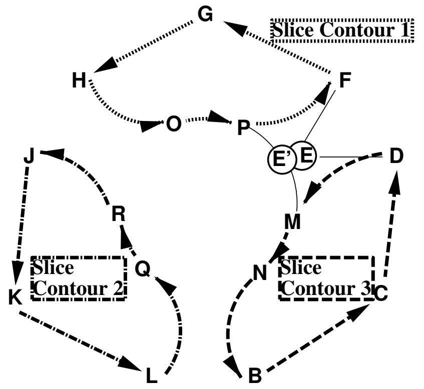

Figure 22: After splicing around the third ending run, one of the splice contours splits again.

were upside down. With this part, an outer contour splits into additional outer contours, but a split can also result in a change of contour orientation, as in the part shown in figure 23, where an outer contour splits into one inner and one outer contour.

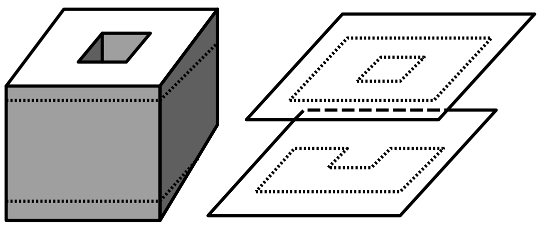

Figure 23: When this part is sliced, an outer contour (bottom right) splits into one inner and one outer contour (top right).

4.3 Contour Classification and Nesting

Before we determine how contours are nested, we classify them as outer contours or inner hole contours, based on whether they are oriented counterclockwise or clockwise as viewed from above. As mentioned previously, we wait until we get to the first slice after the contour is formed to classify it, since we'll be making intersection point calculations here anyhow. We use the ordered intersection points to find the z-component of the contour's normal [8]; positive indicates a counter-clockwise contour, negative a clockwise contour. This classification will remain valid for the contour until it splits or merges, through any number of inserted or deleted edge_uses that don't change its topology.

If we determine that we have an inner hole contour, we must find the outer contour that contains it (the contour within which it subtracts a hole). To optimize for the common case, as well as to reduce the problem size for the uncommon case of many complex outer contours, the first pass compares the bounding box of an inner hole contour with the bounding boxes of all outer contours. If it is contained in the bounding box of only a single outer contour, then that is its containing contour. If it is contained in the bounding boxes of multiple outer contours, we use a ray test to determine which of these is the containing contour. If we have any hole contours, then we must also determine if any of our outer contours are islands within the holes, and nest them appropriately as well. Nesting, like orientation, remains unchanged as we process vertex disk cycles, inserting and deleting edge_uses, until a contour splits or merges, at which time we mark its nesting invalid and rederive it at the next physical slice.

4.4 Geometric Slicing

When our sweep plane reaches a height at which we want a physical slice, we need to calculate intersection points for each of the edge_uses in the current slice contour(s). If this is the first intersection with the edge_use, we calculate its slope, find the intersection, and store these values with the edge_use. For subsequent slicing planes that intersect the same edge_use, we look at the slope and the thickness of the slice to find the appropriate deltas in x and y off the last intersection value, add them to it, and record this new intersection value. For SFF processes that use uniform slice thicknesses (virtually all the current commercial processes), we can use the same deltas for each subsequent intersection calculation with the same edge_use; thus we store the deltas with the edge_use, too. In this way, we exploit the geometric as well as the topological coherence of the slices.

One calculation that we do perform on a point by point basis is to eliminate consecutive matching points and hence zero-length edges in the geometric slice contours as we output them. These occur when a slice plane exactly intersects a vertex and we get coincident intersection vertices for each of the beginning edge_uses at this vertex. We simultaneously eliminate zero-area intersection contours from the output if there are only one or two distinct intersection vertices in the contour.

An additional output optimization that would be useful for triangulated input is to omit redundant output points that are already on the segment between their two neighbors. For example, a slice through the middle of a triangulated cube will have eight vertices by default, but only four of these are needed to define the geometry.

5 Analysis

If the number of triangles in the input is n, and the total number of vertices in the output is k, then the total time for calculating intersections and outputting the geometric slices from slice contours will be $O(k+n)$. We may remove up to $O(n)$ redundant vertices from the output, but we will still need to calculate all the intersections to discover which ones to remove. In the worst case, we'll also need to recalculate orientation and nesting for each slice, again $O(k+n)$.

The majority of the steps of the algorithm for building the slice contours are $O(n)$ or $O(nlogn)$. Building the topological data structure is dominated by the time to build hash tables for the edge_uses (to set sibling pointers) and their vertex_uses (to find if a vertex has been seen before and set the appropriate next vertex edge_use pointer); this takes time $O(nlogn)$. Dividing up pseudo-2-manifold edges is linear except for the sort; in the worst case, where virtually all the edge_uses are coincident, this will be $O(nlogn)$. Sorting the vertices to get the correct processing order is again $O(nlogn)$.

Building disk cycles takes time linear in the number of edge_uses to find their connectivity and construct the linked lists, as well as time linear in the number of edge_uses to follow the next vertex edge_use pointers to find the starting edge_uses for additional disk cycles at non-manifold vertices. Identifying all runs of ending edge_uses just requires iterating through each disk cycle, so this is also linear in the number of edge_uses. The total number of splices is limited to one pair of splices per ending edge_use, so this is again linear in the number of edge_uses. Checking for splitting or merging also occurs at most once per ending edge_use. All of these steps are $O(n)$.

The only part of the algorithm that has the potential to be quadratic is resetting the edge_uses' contour membership pointers after splitting or merging. An example worst case is a part of very high genus, with its through-holes lined up vertically over each other, with as many as possible of the other edge_uses that don't define the holes extending from below the lowest hole to above the highest hole, half on each side of the line of holes. With such a geometry, half of the long edge_uses will need to change their contour membership at the bottom of each hole as a single contour splits, and then again at the top of each hole when the two contours merge. Since such a part can be defined with n/c holes and n/c long edge_uses for a small constant c, updating the contour pointers could take total time $O(n^2)$. To accurately prototype such a part, however, we would want to output at least one slice through each hole and one slice between each pair of holes, in which case the update time for the contour pointers would be no more than the $O(k+n)$ time for outputting slices. Parts of such high genus relative to the number of triangles in their boundaries are very rare; for example, the genus 37 part shown in figure 24 had 107,520 input triangles.

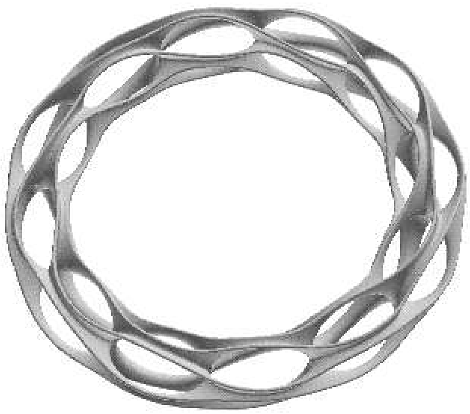

Figure 24: Visualization of the .STL file for a part output from Séquin's sculpture generator [16].

6 Results

We have implemented our algorithm in C++ on an SGI workstation, and tested it on a variety of topologies. The running times reported are on an SGI O2 with two 175 MHZ processors and 128MB main memory.

The preprocessing time to build our topological data structure and check the validity of the input, shown in figure 25, increased at a rate that was slightly more than linear, well within the theoretical bounds of $O(nlogn)$. For a part with 20,000 triangles, the data structure was built in less than ten seconds.

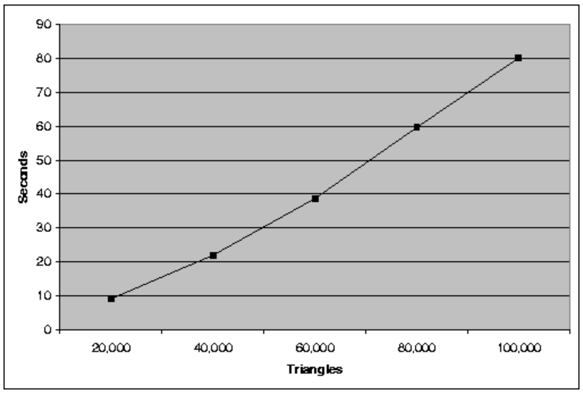

Figure 25: Preprocessing times, including time to build the topological data structure and test for pseudo-2-manifoldness.

The running times for slicing the simple three pronged part of figure 17 are shown in figure 26. All but 27 of the intersection points for this part were calculated incrementally, upwards of 99.9% of the intersections in our tests (with the exception of the comparison test for a single slice).

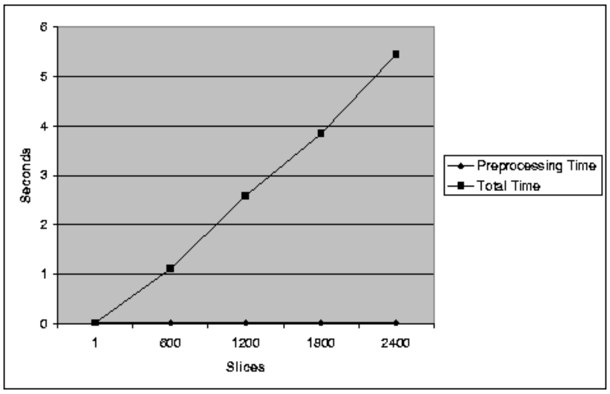

Figure 26: Slicing performance for the three-pronged part (22 input triangles).

The running times for slicing the sculpture pictured in the previous section are shown in figure 27, and the percentage of intersections calculated incrementally are shown in figure 28. Figure 29 illustrates the slice contours our algorithm generated.

7 Future Work

The algorithm as described in this paper works for any input that consists entirely of convex, simple faces, not just triangles. We are in the process of adding the additional functionality needed to address the special cases that can arise when the input faces are concave polygons or contain holes (see figure 30), so that we will be able to slice any non-triangulated polyhedron directly.

To use our algorithm for adaptive layer thicknesses, we would keep track of the minimal slope (closest to horizontal) of any active edge_use and make the distance to the next slice a function of this minimal slope, using thinner slices when it is closer to horizontal. In addition, at vertices with sharp points or horizontal edges with sharp

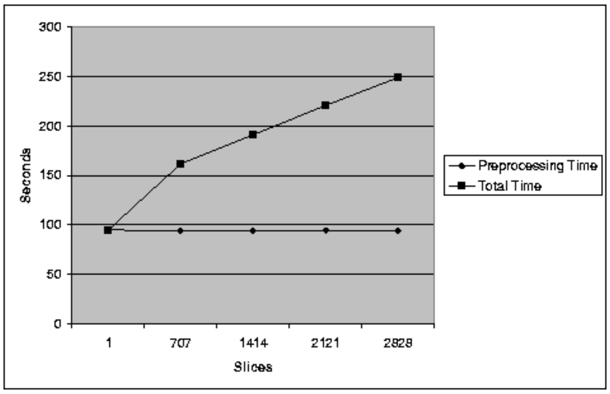

Figure 27: Slicing performance for the sculpture (107,520 input triangles).

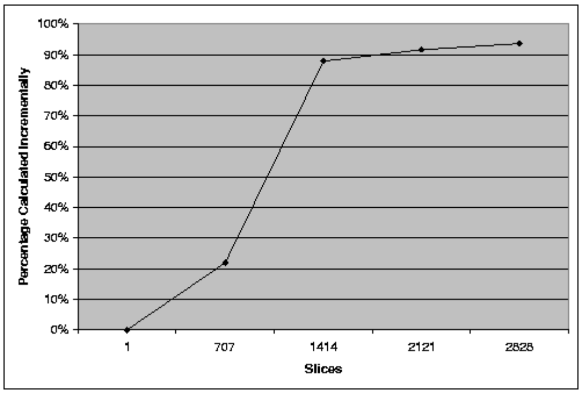

Figure 28: The percentage of intersections calculated incrementally for the sculpture.

dihedral angles, we could introduce an extra slicing plane to capture this extremity.

8 Conclusion

Unlike other slicing algorithms that perform each slice calculation as an individual operation, our algorithm exploits coherence between consecutive slices. Such an approach only makes sense in the case where such coherence actually exists. With most SFF systems, slices must be quite thin, and the number of vertices to be processed between slices is only a small percentage of the total number of edges that intersect a slice. For an application where only a few slices need to be made relative to the number of edges, it would make more sense to use an algorithm that only considered those faces actually cut by the slicing plane, and do the connectivity analysis and matching of edges into contours in 2-D. In general, our sweep plane approach will be most appropriate when $k > n$, i.e. when every polyhedron edge is sliced many times. For the average part, however, the most costly operation is establishing the 3-D connectivity to build the initial topological data structure. This same connectivity analysis is needed to determine if a part description is in fact the boundary of a valid, closed solid, which we would need to verify no matter which slicing algorithm we used. With our algorithm, we derive the connectivity once, verify the part description, and then use the same information for slicing, instead of rederiving the connectivity in individual slices.

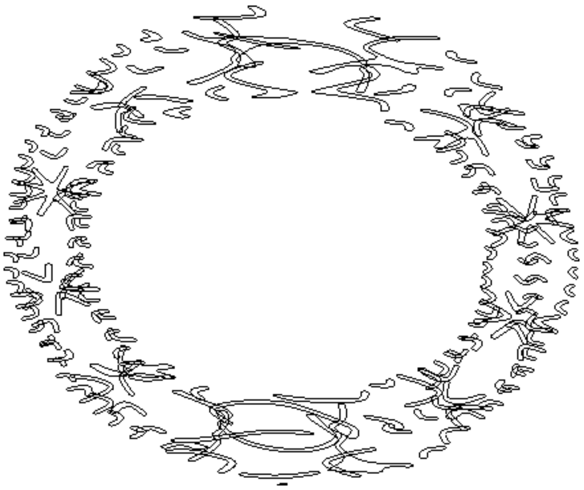

Figure 29: A visualization of the slices through the sculpture. Only a subset of the slices are pictured so that the individual slice contours can be seen clearly.

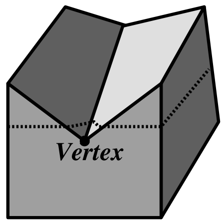

Figure 30: Due to the concavity at the indicated vertex, we don't start a new slice contour even though the vertex's disk cycle contains all beginning edge_uses.

9 Acknowledgements

This paper is taken in part from a thesis to be submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Electrical Engineering and Computer science at U.C. Berkeley. This work was supported by grant MIP-9632345 from the National Science Foundation.

Many thanks to Jordan Smith, who implemented many of the support libraries used in this application, and who suggested that splicing around runs of ending edge_uses instead of runs of beginning edge_uses might make for a simpler, more elegant algorithm. Thanks also to the reviewers for their comments and suggestions.

References

[1] 3D Systems, Inc. "Stereolithography Interface Specification". Company literature, 1988.

[2] Gill Barequet and Micha Sharir. Filling gaps in the boundary of a polyhedron. *Computer-Aided Geometric Design*, 12(2):207–29, March 1995.

[3] B. G. Baumgart. A Polyhedron Representation for Computer Vision. In *Proceedings of the National Computer Conference*, pages 589–596, 1975.

[4] Joseph J. Beaman et al. *Solid Freeform Fabrication : A New Direction in Manufacturing*. Kluwer Academic Publishers, Dordrecht, 1997.

[5] Jan Helge Bohn and Michael J. Wozny. A Topology-Based Approach for Shell-Closure. In *Geometric Modeling for Product Realization*, pages 297–319. North-Holland, Amsterdam, 1992.

[6] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, 1997.

[7] A. Dolenc and I. Mäkelä. Slicing procedures for layered manufacturing techniques. *Computer Aided Design*, 26(2):119–26, February 1994.

[8] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*, page 477. Addison-Wesley, Reading, MA, 2nd edition, 1990.

[9] E. Gursoz, Y. Choi, and F. Prinz. Vertex-based Representation of Non-manifold Boundaries. In *Geometric Modeling for Product Engineering*, pages 107–130. North-Holland, Amsterdam, 1990.

[10] C. F. Kirschman and C. C. Jara-Almonte. A Parallel slicing algorithm for solid freeform fabrication processes. In *Proceedings of the Solid Freeform Fabrication Symposium*. University of Texas at Austin, August 1992.

[11] Prashant Kulkarni and Debasish Dutta. An Accurate Slicing Procedure for Layered Manufacturing. *Computer Aided Design*, 28(9):683–97, September 1996.

[12] Martti Mäntylä. Boolean Operations of 2-Manifolds through Vertex Neighborhood Classification. *ACM Transactions on Graphics*, 5(1):1–29, 1986.

[13] Martti Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.

[14] T. M. Murali and Thomas A. Funkhouser. Consistent solid and boundary representations from arbitrary polygonal data. In *1997 Symposium on Interactive 3D Graphics*, pages 155–162, Providence, R.I., April 1997. ACM SIGGRAPH.

[15] Stephen J. Rock and Michael J. Wozny. Utilizing Topological Information to Increase Scan Vector Generation Efficiency. *Solid Freeform Fabrication Symposium Proceedings*, pages 28–36, 1991.

[16] C. H. Séquin, H. Meshkin, and L. Downs. "Interactive Generation of Scherk-Collins Sculptures". In *1997 Symposium on Interactive 3D Graphics*, pages 163–166, Providence, RI, April 27-30 1997. ACM Siggraph.

[17] Spatial Technology, Inc, Boulder, CO. *ACIS Save File Format Manual*, 1996.

[18] H. Voelcker. "Modeling in the Design Process". In *Design and Analysis of Integrated Manufacturing Systems*, pages 167–199. National Academy Press, Washington, DC, 1988.

[19] Kevin Weiler. The Radial Edge Structure: a Topological Representation for Non-manifold Geometric Boundary Modeling. In *Geometric Modeling for CAD Applications*, pages 3–36. North-Holland, Amsterdam, 1988.